Bachelorthesis

Migration der Nutzerwelten RCP Anwendung von Eclipse Version 3.7 nach Eclipse 4.x

Autor: Stephen Schulte Mat:524592

Betreuer: Prof. Dr. rer. nat. Wolfgang Lux
Zweitbetreuer: Prof.Dr.-Ing.Ulrich G. Schaarschmidt

Im Zeitraum vom 18.11.2014 bis zum 10.02.2015

Fachhochschule Düsseldorf
Fachbereich: Elektrotechnik
Studiengang: Informationstechnik

Inhaltsverzeichnis

Seite

Eidesstattliche Erklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Alle Ausführungen, die fremden Quellen wörtlich oder sinngemäß entnommen wurden, sind kenntlich gemacht. Die Arbeit war in gleicher oder ähnlicher Form noch nicht Bestandteil einer Studien- oder Prüfungsleistung.

Düsseldorf, den 20.11.2014

Stephen Schulte

Danksagung

Hiermit möchte ich mich bei allen herzlich bedanken, die mich während der Anfertigung meiner Bachelorarbeit unterstützt haben.
Speziell gilt mein Dank...

- Prof. Dr. rer. nat. Wolfgang Lux für die Bereitstellung des Themas sowie für die angenehme Betreuung.

- Oliver von Fragstein für die ständige Unterstützung innerhalb des Labors, die vielen hilfreichen Tipps und die geduldige und unkomplizierte Beantwortung meiner Fragen.

- Leandro F. Rojas Peña für das Beantworten meiner Fragen zu den Themen Java und Eclipse.

Düsseldorf, den 04.01.2014
Stephen Schulte

Zusammenfassung

Im Rahmen dieser Bachelorthesis sollen die Möglichkeiten für eine Migration des Nutzerwelten Programmcodes in die neuen Eclipse 4 Versionen ermittelt und anhand von Beispielen schrittweise erklärt werden. Hierbei werden zunächst die neuen Möglichkeiten erläutert die eine Eclipse 4 Entwicklung bietet und welche Vorteile durch die Einführung von Dependency Injections und dem Application Model entstanden sind. Anschließend werden die möglichen Wege zur Durchführung einer Migration aufgezeigt und auf ihre Vorteile und Nachteile überprüft. Im letzten Abschnitt werden schließlich die Migrationswege anhand von Beispielen schrittweise an einem Bundle der Nutzerwelten RCP durchgeführt und auf ihren Nutzen für eine Migration des gesamten Programms bewertet.

Abstract

In the context of this bachelor thesis the possibilities for an migration of the Nutzerwelten program code should be determined by using examples and migrating the code step by step. At first we take a look at the new feature of the Eclipse 4 development and which advantages are given by using dependency injections and the application model. Afterwards the possible ways for an migration are demonstrated and verified for there positive and negative aspects. Finally the possible migration ways are shown at an example migration of the Nutzerwelten RCP bundle and the utility for an migration of the whole program is checked.

Projektvorstellung

1.1 Die Ausgangslage

Die Nutzerwelten RCP besteht aus einer Reihe von Bundles, die über die Jahre in verschiedenen Projekten zum bestehenden Programm hinzugefügt wurden. Das Projekt wurde seit der Entstehung konstant ausgebaut und aktualisiert, um die Bedürfnisse der Kunden bestmöglich abzudecken. Aufgrund des längeren Entwicklungszeitraums des WieDAS Projekts ist die Eclipse Version mit der die Programmierung durchgeführt wurde nicht mehr auf dem aktuellen Stand. Aufgabe dieser Bachelorarbeit ist es, die Möglichkeiten für eine Migration der Nutzerwelten RCP in die neue Eclipse 4 Version zu ermitteln und die Schritte für eine erfolgreiche Migration aufzuzeigen.

1.2 Das WieDAS Projekt

An der Fachhochschule in Düsseldorf und an der Hochschule RheinMain wurde im Zeitraum vom 1.7.2010 bis zum 30.09.2013 ein AAL Projekt mit dem Namen WieDAS entwickelt. Das WieDAS Projekt beschäftigt sich mit der Entwicklung und dem Einsatz von Geräten und Software die es älteren Menschen erlauben länger in ihrer gewohnten Umgebung zu verbleiben.

Eine ausführliche Beschreibung des WieDAS Projekts, sowie der dort eingesetzten Geräte und der anschließend durchgeführten Studie, befindet sich im Praxisprojekt *"Entwickeln und Einbinden einer Fensterkontrollsoftware im Rahmen des AAL Projektes WieDAS".*

Nach Abschluss des WieDAS Projekts wurde die Software und die zur Verfügung stehenden Geräte auf die Bedürfnisse der Testpersonen angepasst und ein neues Projekt mit dem Namen Nutzerwelten gestartet. Für das Projekt wurden einige neue Geräte zu der vorhanden Auswahl hinzugefügt.

1.3 Das Nutzerwelten Projekt [7]

Für das Projekt Nutzerwelten wurde der aktuelle Stand des WieDAS Projekts eingefroren um einen festen Projektstand für die weitere Entwicklung vorliegen zu haben. Für die weitere Bearbeitung des Projekts innerhalb der Fachhochschule haben sich weitere Fachbereiche an der Weiterentwicklung beteiligt.

Der Fachbereich Informatik übernimmt weiterhin die Entwicklung der Geräte und von deren Software. Das Hauptaugenmerk des Fachbereichs Informatik liegt beim Nutzerwelten Projekt nicht auf der Entwicklung neuer Geräte, sondern auf das Herstellen und Einbauen weiterer Hardware, um eine möglichst große Anzahl von Testhaushalten zu versorgen.

Der Fachbereich Soziologie beschäftigt sich mit den sozialen Schwerpunkten des Projekts und führt Umfragen zur Lebensqualität und zu Anforderungen von älteren Menschen an ein Assistenzsystem durch.

Der Fachbereich Design arbeitet an der optischen und der akustischen Gestaltung der Geräte, um eine möglichst hohe Akzeptanz der Zielpersonen zu gewährleisten.

Das Ziel von Nutzerwelten ist der Aufbau eines Netzwerks mit nutzerorientierter Entwicklung. Hierfür wird das Projekt in dynamischen Bausteinen entwickelt und kann auf jeden Nutzer, je nach individuellen Bedürfnissen, angepasst werden. Die fertig entwickelten Geräte werden anschließend in Testhaushalten installiert und auf ihre Funktionsfähigkeit überprüft. Hierbei ist ein fehlerfreier Ablauf der Software und Hardware von großer Bedeutung. Ebenso ist es von Bedeutung wie wichtig den Testpersonen der Einsatz der verschiedenen Geräte erscheint.

1.3.1 Bestandteile der Nutzerwelten RCP [7]

Die Nutzerwelten RCP umfasst die folgenden Sieben Geräte:

- Herdüberwachung
- Bewegungsmelder
- Sturzerkennung
- Temperatursensor
- Wassermelder
- Steckdose
- Fensterdetektor

Die im Nutzerwelten Projekt eingesetzten Geräte sind hierbei in verschiedene Bereiche unterteilt. Die Hardware beschreibt den eigentlichen Sensor, welcher die Daten (beispielsweise Temperatur) aufnimmt und mittels IP-Connector an das logische Gerät weitersendet. Diese Informationen werden vom logischen Gerät (Device) verarbeitet und an die View weitergeschickt. In der View werden die Messungen der Sensoren anschließend grafisch dargestellt.

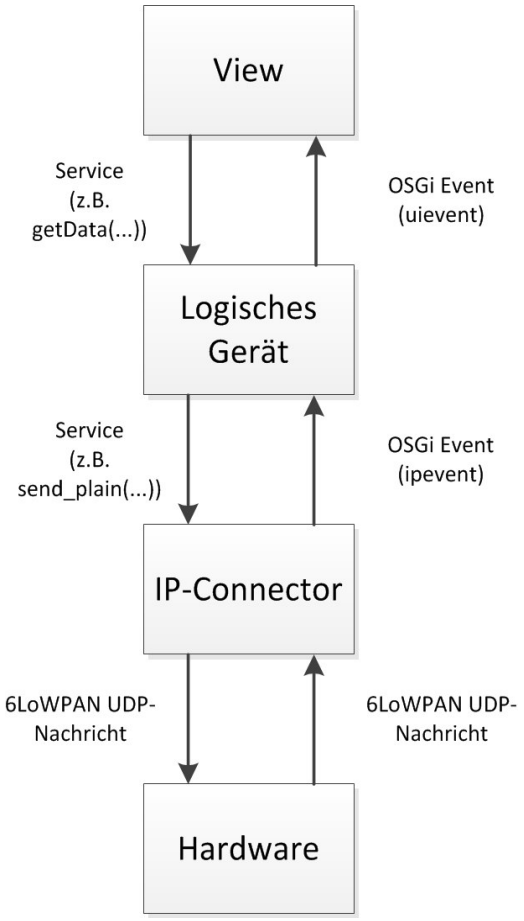


Bild: 1 OSGI-Bundle Struktur des Informatiklabors

Die Entwicklung der Nutzerwelten RCP wurde mit der dynamischen Service Platform OSGI realisiert. Durch die Entwicklung mit OSGI ist es leicht möglich zusätzliche Geräte zum Programm hinzuzufügen. Ebenso leicht können Geräte aus dem Programm entfernt werden, wenn diese bei einem Nutzer nicht benötigt werden.

Eine Einführung zur Service Platform OSGI, sowie der Beispielhafte Aufbau eines Bundles innerhalb der Nutzerwelten RCP, befindet sich im Praxisprojekt *"Entwickeln und Einbinden einer Fensterkontrollsoftware im Rahmen des AAL Projektes WieDAS".*

Eclipse

2.1 Was ist Eclipse? [10]

Bei Eclipse handelt es sich um ein plattformunabhängiges Programmierwerkzeug (Editor) mit dem Programme in verschiedenen Programmiersprachen erstellt werden können. Das Hauptaugenmerk von Eclipse lag bei seiner Gründung in der Firma IBM auf der Programmiersprache Java[11]. Am 2 Februar 2004 wurde das von IBM geführte Eclipse Konsortium selbstständig und ist seitdem unter dem Namen Eclipse Foundation für die Entwicklung von Eclipse verantwortlich. Seit am 28 Juni 2004 die Version 3.0 veröffentlicht wurde, erscheint in jedem Jahr eine neue Eclipse Version. Die WieDAS / Nutzerwelten RCP wurde in der am 22.Juni.2011 erschienenen Version 3.7 (Indigo) verfasst.

2.1.1 Eclipse IDE [5]

Die Eclipse IDE (Integrated Development Environment) war ursprünglich nicht als Plattform zur Programmentwicklung entwickelt worden. Stattdessen sollte Eclipse eine Plattform bieten auf der Kompatibilitätsprobleme anderer Programmier-werkzeuge mithilfe von Plug-ins gelöst werden können. Die Eclipse IDE hat sich aufgrund der Flexibilität des Plug-in Aufbaus als Entwicklungsoberfläche durchgesetzt und ist mittlerweile die meistgenutzte IDE weltweit. Die Eclipse IDE liefert die grafische Entwicklungsoberfläche und bietet die Möglichkeit zusätzliche Funktionen durch OSGI in Form von Plug-ins in die Eclipse IDE zu laden. Gerade durch die Möglichkeit die benötigten Bestandteile individuell in die Eclipse IDE einbinden zu können, wird das große Potenzial ersichtlich. Die wichtigste Erweiterungsmöglichkeit der IDE besteht darin Plug-ins für weitere Sprachen hinzufügen zu können[11]. Das C und C++ Plug-in Developer Tools (CDT) und das Dynamic Language Toolkit Project (DLTK) sind zwei mögliche Bundles zum Integrieren neuer Sprachen. Weitere Bestandteile der Eclipse IDE sind ein integrierter Update Manager und die Hilfskomponenten.
2004 wurden diese grundlegenden Bestandteile der Eclipse IDE extrahiert und als Eclipse RCP veröffentlicht. Diese Bestandteile können nun auch unabhängig benutzt werden.

2.1.2 Eclipse RCP [5]

Die Eclipse RCP (Rich Client Platform) bezeichnet eine Ansammlung von Bundles und Programmen aller grundlegenden Bausteine die benötigt werden um eine lauffähige Anwendung mit grafischer Ausgabe zu gestalten. Die grafische Darstellung erfolgt über das Plug-in SWT. Das Anbieten der einzelnen Bauteile der Eclipse RCP wird mithilfe von Plug-Ins über OSGI realisiert. OSGI ermöglicht das flexible Hinzufügen und Entfernen von Plug-Ins und erlaubt es dem Nutzer nur die Bestandteile der RCP zu laden die genutzt werden sollen. Somit kann der Nutzer auch weitere Bestandteile zu seiner RCP hinzufügen und diese nach seinen eigenen Bedürfnissen gestalten.

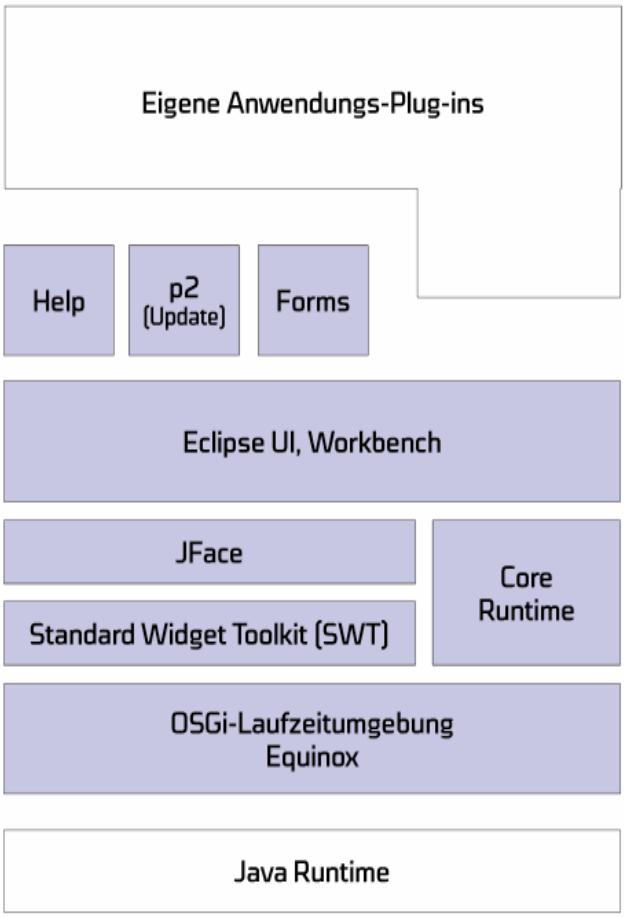


Bild: 2 Aufbau der Eclipse RCP (Version 3.7)

In Eclipse 4 ist zusätzlich zu den Funktionen der Eclipse 3 RCP noch eine Kompatibilitätsschicht enthalten. Diese ermöglicht es Programmcode, der in Eclipse 3 verfasst wurde, auch in Eclipse 4 zu verwenden. Des Weiteren unterstützt die Eclipse 4 RCP auch das Verwenden von CSS Dateien.

Die folgenden Bundles gehören zusätzlich zu den in der Grafik gezeigten Eclipse 3 Plug-Ins zum Standartbaukasten bei einer Eclipse 4 RCP. [13]

- Dependency Injection
- IEclipseContext
- Core Services
- Application Model
- Rendering Engine
- Declarative Styling.

Außerdem gibt es noch Bausteine die nicht im offiziellen Eclipse 4 Release enthalten sind, aber optional hinzugefügt werden können. Zu diesen optionalen Bauteilen gehört beispielsweise XWT. Mit XWT können SWT Oberflächen mit XML beschrieben werden.

2.2 Eclipse Version 4 [10]

Die Entwicklungsgewohnheiten der Nutzer haben sich seit der Einführung von Eclipse stark verändert. Immer häufiger werden Projekte verteilt an vielen Standpunkten auf der Welt gleichzeitig entwickelt. Des Weiteren haben sich bestimmte Neuerungen, die bislang nur über Plug-ins verfügbar waren, bei den Entwicklern durchgesetzt, wie beispielsweise die Nutzung von Dependency Injections. All dies führte dazu, das mit dem Eclipse e4 Projekt schließlich der Grundstein für eine neue Eclipse Generation gelegt wurde. Am 27.Juni 2012 ist mit Eclipse Juno die erste Eclipse Version der 4ten Generation erschienen. Eclipse 4.2 läuft zu diesem Zeitpunkt parallel mit der Version 3.8, die Fehler aus der Version 3.7 beheben soll. Als im Folgejahr Eclipse Kepler erscheint, wird Eclipse 4 als offizieller Nachfolger von Eclipse 3 festgelegt und alle zukünftigen Projekte sollen in Eclipse 4 entwickelt werden.
Die wichtigsten Neuerungen die in Eclipse 4 eingeführt wurden, werden im folgenden Kapitel vorgestellt.

2.2.1 Dependency Injections [3]

Um in Eclipse 3 Abhängigkeiten aufzulösen, wird oft eine Vielzahl von sogenannten Singletons (Entwurfsmuster) verwendet. Diese Singletons ermöglichen es sicherzustellen, dass von einer Klasse nur genau ein Objekt existiert. Durch eine exzessive Verwendung von Singletons wird allerdings ein Äquivalent von globalen Variablen erschaffen, die in einer objektorientierten Programmierung nicht genutzt werden sollten. Eine Möglichkeit diese Singletons zu ersetzen und die Struktur des Programmcodes zu vereinfachen bieten die sogenannten Dependency Injections.

In Eclipse 4 wurde die Möglichkeit zur Nutzung von Dependency Injections in das Standard Framework integriert und ist seitdem fest im Framework verankert. Andere Möglichkeiten an seine Services oder sonstige Informationen zu kommen, werden absichtlich nicht länger unterstützt (Eine Verwendung der Singletons soll unterbunden werden). In früheren Versionen von Eclipse war das Verwenden von Dependency Injections nur möglich, wenn im Workbench externe Plug-ins hinzugefügt wurden.

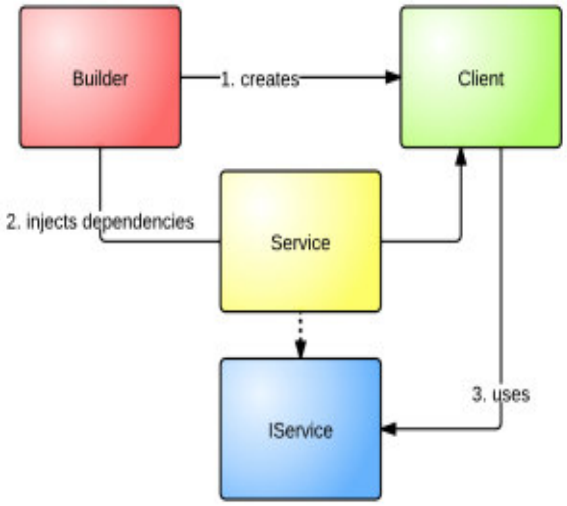


Bild: 3 Dependency Injections

Dependency Injections ermöglichen es Klassen auf andere von ihnen benötigte Objekte zuzugreifen. Das Ziel hierbei ist es die Anzahl benötigter Abhängigkeiten zu minimieren und den Code möglichst übersichtlich zu gestalten. Dies geschieht indem die Abhängigkeiten nicht mehr innerhalb der Klasse festgelegt werden. Stattdessen zeigen verschiedene Schlüsselwörter, wie beispielsweise das Schlüsselwort *@Inject* an, dass an dieser Stelle Objekte von außerhalb benötigt werden. Bei Eclipse 4 Versionen werden die benötigten Objekte von einem "Eclipse Context" verwaltet. Im Eclipse Context werden die zur Verfügung stehenden Services angemeldet. Anschließend überprüft der Eclipse Context den Code nach für ihn relevanten Schlüsselwörtern und fügt die benötigten Objekte ein. Mit diesem Verfahren können harte Abhängigkeiten zu anderen Klassen aufgelöst werden und Testläufe einzelner Klassen werden erleichtert.

Vorteile von Dependency Injections [14]

Reduzieren von Anhängigkeiten

Die Anzahl von überflüssigen Abhängigkeiten kann durch das Verwenden von Dependency Injections reduziert werden, oder in manchen Fällen sogar ganz entfernt werden. Ein Beispiel für das Entfernen einer überflüssigen Abhängigkeit ist es, wenn eine Klasse nur deswegen implementiert werden muss, weil diese von einer anderen Klasse benötigt wird. Diese Verkettung von Abhängigkeiten wird durch das Anlegen als Services umgangen.

Bessere Testmöglichkeiten

Mit Dependency Injections können Klassen getestet werden ohne das Klassen, zu denen sonst eine harte Abhängigkeit vorliegen würde, existieren müssen. Mithilfe des *new* Operators wird hierbei ein Dummy von der abhängigen Klasse zu Testzwecken angelegt. Die Komponenten der Dummy Klasse können angepasst werden um den Test mit verschiedenen übertragenen Parametern durchzuführen.

Einfacher Austausch von Programmteilen

Durch Dependency Injections ist es leichter Programmteile auszutauschen, da diese nicht mehr direkt voneinander abhängig sind. Der Verweis um die benötigten Objekte zu bekommen findet über das Interface statt und nicht länger über die Klasse selbst.

Bessere Lesbarkeit

Durch das Verschieben der Abhängigkeiten in das Interface lässt sich leichter erkennen welche Abhängigkeiten eine Klasse besitzt. Um die Abhängigkeiten der verschiedenen Klassen zu überprüfen, müssen diese nicht länger einzeln aufgerufen werden. Die Abhängigkeiten können alle im Interface angezeigt werden.

Eclipse Context [1]

Der Eclipse Context ist für das Suchen und Finden der angelegten Services zuständig. Er verwaltet alle Objekte die in Klassen integriert werden können und legt für diese eine Liste an. Die Objekte werden hierbei unter ihrem vollständigen Klassennamen abgespeichert. Um stets auf das richtige Objekt Zugriff zu erlangen, gibt es in Eclipse mehrere Contexte für die Verwaltung von Objekten. Diese Contexte sind in einer hierarchischen Reihenfolge aufgebaut und werden bei der Suche nach dem zu injizierenden Objekt von oben nach unten abgesucht. Bei der Suche nach einem Objekt wird zunächst ein lokaler Context aufgerufen der die eigene Klasse nach einem passenden Objekt durchsucht. Wenn dieser keinen passenden Eintrag finden kann, wird die Suche ausgeweitet. Die letzte Instanz stellt der OSGI Context dar, der die gesamte Anwendung nach einem passenden Objekt durchsucht.
Der Context beinhaltet alle Elemente des Application Models. So kann man sich beispielsweise aus dem Context eines Parts das Fenster injizieren lassen in dem der Part enthalten ist. Mit dem Application Model assoziierte SWT Elemente wie beispielsweise Composites oder Parts können ebenfalls über den Context abgerufen werden [2]. Ebenso ist es möglich eigene Objekte in den Context einzufügen.

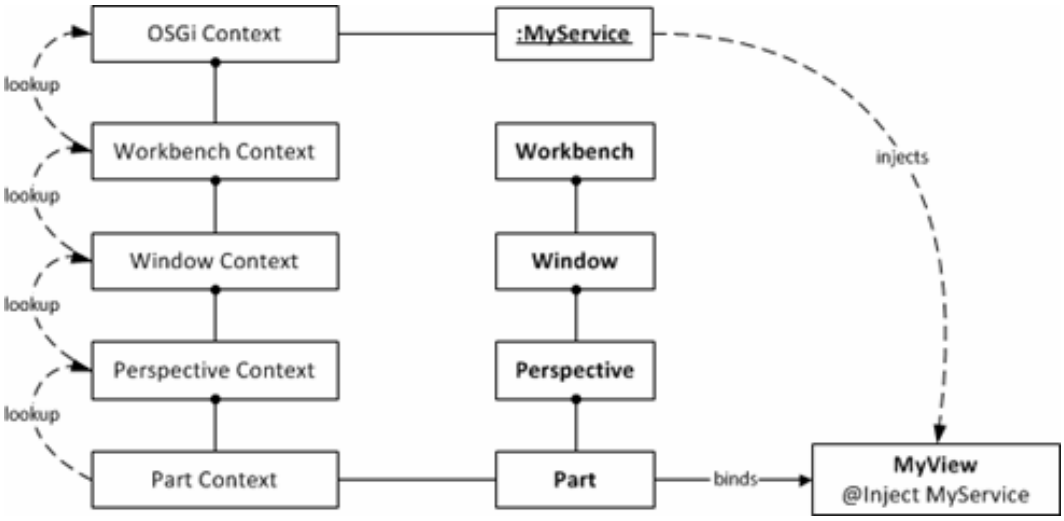


Bild: 4 Aufbau des Eclipse Context

Dependency Injections Schlüsselwörter [2]

Bei Dependency Injections können verschiedene Schlüsselwörter für das Injizieren der Daten verwendet werden. Durch das Verwenden unterschiedlicher Schlüsselwörter kann der Zeitpunkt des Injizierens verändert werden, oder spezifischer nach einem genauen Objekt gesucht werden. Eine genauere Suche wird immer dann benötigt, wenn mehrere Objekte auf die Suchanfrage zutreffen und die Suche somit weiter spezifiziert werden muss (siehe method injection). Einige der häufig verwendeten Schlüsselwörter werden hier kurz vorgestellt.

@Inject

@Inject ist das Standard Schlüsselwort um anzuzeigen dass jetzt eine Dependency Injection durchgeführt werden soll. Das Schlüsselwort kann mit anderen Schlüsselwörtern erweitert werden. Im Fall das kein passendes Objekt zum Einfügen gefunden wird, wird das Programm mit einer Fehlermeldung beendet.

@Named

Das Schlüsselwort *@Named* kann zusätzlich zu *@Inject* verwendet werden und erlaubt eine zusätzliche Spezialisierung des Objekts, da nun der Name des einzufügenden Objekts in Form eines zusätzlichen Strings festgelegt wird. Dieses Vorgehen ermöglicht es nicht nur einen bestimmten Typ, sondern auf eine ganze Instanz dieses Typs zuzugreifen. Objekte können innerhalb des Eclipse Context auch mit einem vorgegeben Namen abgespeichert werden. Diese Objekte können ebenfalls mit dem Schlüsselwort *@Named* abgerufen werden.

@Optional

Mit dem Schlüsselwort *@Optional* wird ebenso wie bei *@Inject* ein passendes Objekt zum Einfügen gesucht. Falls kein passendes Objekt vorhanden ist, wird anders als bei *@Inject* jedoch keine Fehlermeldung ausgegeben und das Programm anschließend beendet. Stattdessen wird das gesuchte Objekt mit einem Nullwert gefüllt.

@Active

@Active wird verwendet wenn es nötig ist das zu diesem Zeitpunkt aktive Objekt einzufügen, anstatt nur eines beliebigen Elements. Durch dieses Kommando können beispielsweise Handler auf den zurzeit aktiven Part zugreifen.

@PostConstruct und @PreDestroy

Bei dem Befehl @Postconstruct werden die benötigten Objekte erst eingefügt nachdem alle Objekte komplett injiziert sind. Dies ist besonders dadurch relevant da aufgrund der Reihenfolge der Dependency Injection im Konstruktor noch nicht auf die Felder einer Klasse zugegriffen werden kann. @Predestroy wird benutzt um ein Objekt wieder abzumelden, wenn es nicht mehr benötigt wird. Für beide Kommandos können zusätzliche Parameter angewendet werden, müssen aber nicht.

Injektionen durchführen [2]

Um eine Dependency Injection im Programmcode zu verwenden, wird das Schlüsselwort *@Inject* mit eventuell benötigten zusätzlichen Kommandos vor den Programmteil geschrieben der eine Injektion erhalten soll. Die möglichen Programmteile, die eine Injektion erhalten können, sind Konstruktoren, Felder oder Methoden. Im Falle mehrerer durchzuführender Injektionen spielt die Reihenfolge der Injektionen eine große Rolle. Als erstes wird die Injektion von Konstruktoren durchgeführt. Anschließend wird die Injektion von Feldern vorgenommen. Abschließend werden die Methoden injiziert.
Sollten sich die im Context gespeicherten Objekte während der Durchführung ändern, wird die Injektion erneut durchgeführt.

Konstruktoren: (construction injection)

In einem Konstruktor sollen alle Parameter enthalten sein die für ein Objekt von essenzieller Bedeutung sind. Hierbei ist es wichtig, möglichst nur die benötigten Parameter anzugeben, da jedes unnötige Parameter die Wiederverwendbarkeit und die Testbarkeit des Objektes einschränkt.
Ein typisches Beispiel für eine Dependency Injection bei einem Konstruktor ist das Injizieren des Parents für einen Part. Da die View im Context eines Parts aus dem Application Model initialisiert wird, ist die Angabe des Typs *Composite* in diesem Fall eindeutig.

```
@inject
public void MyView(composite parent){
}
```

Felder: (field injection)

Eine typische Anwendung für das Injizieren eines Feldes ist das Injizieren eines Services der in der Klasse global zur Verfügung gestellt werden soll. Da Services im Normalfall nur einmal pro Anwendung vorkommen, reicht auch hier die Angabe von einem Typ.
Bei der Injektion von Feldern ist darüber hinaus zu beachten, dass diese nicht mit dem Typ `final` deklariert sind. Felder mit dem Typ `final` müssen explizit über den Konstruktor gesetzt und dort injiziert werden.

Methoden: (method injection)

Im Anschluss an die Konstruktoren und die Felder werden die Methoden injiziert. Ein Beispiel für das Injizieren einer Methode ist die aktuelle Selection, auf die man häufig mit einer View oder einem Handler reagieren möchte. Da es mehrere Objekte mit passenden Parametern gibt, reicht es in diesem Fall nicht aus, nur das Schlüsselwort *@Inject* zu verwenden. Zusätzlich muss die Annotation *@Named* hinzugefügt werden, um das Suchobjekt weiter zu spezifizieren. Außerdem sollte das Schlüsselwort *@Optional* hinzugefügt werden. Da beim Start der Anwendung noch keine Selection im Context vorhanden ist, würde hier eine Fehlermeldung erscheinen.

Beispiel für die Umkehr von Abhängigkeiten [14]

Die größte Aufgabe von Dependency Injection ist es, die Abhängigkeiten von Objekten während der Laufzeit zu reglementieren. Hierbei wird die Abhängigkeit nicht länger von der Klasse selbst übernommen, sondern in einem zentralen Ort hinterlegt. Dieses Vorgehen erleichtert die Testläufe von Klassen und erhöht die Wiederverwendbarkeit des Codes.
Diese *"Inversion of Control"* (Umkehr der Abhängigkeiten) soll hier anhand des Beispiels an einem Konstruktor erklärt werden.

Zunächst wird die Klasse MyDao (Data Access Object) gezeigt, ohne den Einsatz von Dependency Injections.

```
public class MyDao {

    protected DataSource dataSource =
    new DataSourceImpl("driver", "url", "user", "password");

    //data access methods...
    public Person readPerson(int primaryKey) {...}

  }
```

Um zu funktionieren benötigt die Klasse MyDao eine Instanz des Typs DataSourceImpl für den Zugriff auf die Datenbank. Ohne eine solche Instanz ist der Programmcode nicht lauffähig. Eine solche Abhängigkeit wird daher als "harte" Abhängigkeit bezeichnet. Eine harte Abhängigkeit hat auch zur Folge, dass der Austausch von Abhängigkeiten erschwert wird. In diesem Fall muss die Klasse MyDao umgeschrieben werden, wenn für das Einlesen der Daten eine andere Datenbank verwendet werden soll. Wenn ein Programm über mehrere Klassen verfügt, die Daten aus dieser Datenbank auslesen, muss jede Einzelne dieser Klassen umgeschrieben werden.

Um die Klasse unabhängiger zu gestalten, wird der Code nun angepasst.

```
public class MyDao {

  protected DataSource dataSource = null;

  public MyDao(String driver, String url, String user, String password){
    this.dataSource = new DataSourceImpl(driver, url, user, password);
  }

  //data access methods...
  public Person readPerson(int primaryKey) {...}

}
```

Bei dieser Variante der Klasse wird die DataSourceImpl Initialisierung in einen Konstruktor verschoben. Der Konstruktor nimmt hierbei die vier Parameter die von der DataSource Implementierung benötigt werden.
Die Klasse ist dadurch zwar immer noch von diesen vier Werten abhängig, allerdings werden diese Abhängigkeiten nicht länger von der Klasse selbst zur Verfügung gestellt. Stattdessen werden diese Werte von der Klasse, welche die MyDao Instanz zur Verfügung stellt, injiziert. In dieser Ausführung des Codes ist es möglich, die Datenbanken für driver, url, user und password zu ändern, ohne dabei eine Änderung am Code der MyDao Klasse vornehmen zu müssen.

Um die Klasse noch unabhängiger zu gestalten, kann der Code erneut angepasst werden.

```
public class MyDao {

    protected DataSource dataSource = null;

    public MyDao(DataSource dataSource){
      this.dataSource = dataSource;
    }

    //data access methods...
    public Person readPerson(int primaryKey) {...}

  }
```

In dieser Variante ist die Klasse nicht länger von der Klasse DataSourceImpl oder den vier Strings abhängig. Für die Injection in diese Klasse kann nun eine beliebige DataSource Implementierung innerhalb des Konstruktors verwendet werden.

2.2.2 Das Application Model [1,2]

Um die Struktur eines Programms in Eclipse 4 darzustellen, wird eine abstrakte Beschreibungsform mit dem Namen Application Model verwendet. Für jede Eclipse 4 Anwendung wird ein eigenes Application Model angelegt. Innerhalb dieses Modells werden alle visuellen, sowie einige ausgewählte nicht visuelle Bestandteile, des Programms aufgeführt. Das Application Model wird komplett mit dem Eclipse Modeling Tool aufgebaut und soll das Konzept weiterverfolgen alle Anwendungen deklarativ mit XML zu beschreiben. Eine Arbeit mit dem Application Model ist bereits möglich, bevor die einzelnen Komponenten angelegt wurden. Beispielsweise kann mit dem Application Model gearbeitet werden, bevor eine View angelegt wurde.

Sichtbare Bestandteile:	windows,views,menus und toolbars.
Nicht sichtbare Bestandteile:	handlers,commands und key bindings.

In einem Application Model werden alle vorhandenen Bestandteile und die Struktur der Anwendungen beschrieben, allerdings nicht der Inhalt der jeweiligen Anwendungen. Wenn beispielsweise als Objekt ein Label anlegt wird, würde dieses als Label im Application Model mit allen seinen Eigenschaften angezeigt werden. Allerdings würde der im Label eingetragene Text nicht im Application Model erscheinen.

Das Application Model kann während der Laufzeit angepasst werden und ermöglicht somit eine dynamische Arbeitsweise. Der Zugriff auf das Application Model erfolgt mit dem Kommando EModelService. Wenn während der Laufzeit durch diesen Befehl die Größe eines Fensters verändert wird, geschieht dies auch auf der Nutzeroberfläche. Das Application Model ist Tool unabhängig und ermöglicht es ohne Änderungen auch Programme auf der Basis von beispielsweise Swing darzustellen.
Eine weitere Möglichkeit das Application Model während der Laufzeit anzupassen bietet das zusätzliche Werkzeug "Live Model Editor", welches über das Plug-In `"org.eclipse.e4.tools.emf.liveeditor"` hinzugefügt werden kann. Mit diesem Tool ist es möglich den XMI Code des laufenden Programms auszulesen. Diese Methode wird im Kapitel "*Eine reine Eclipse Migration*" verwendet um das Implementieren des Application Models zu vereinfachen.

Um festzulegen welches Modell ein Application Model Erzeugen soll, muss das Application Model als Extensionpoint deklariert werden. Beim Erzeugen des Extension Points wird die Datei plugin.xml automatisch erzeugt, wenn sie noch nicht vorhanden ist. Das folgende Kommando zeigt die Deklaration eines Application Models in der Plug-In Datei.

```
<product
            name="com.example.e4.rcp.wizard"
            application="org.eclipse.e4.ui.workbench.swt.E4Application">
</product>
```

Aufbau des Application Model [1]

Das Application Model wird im Plug-In durch eine Datei mit der Endung *.e4xmi* dargestellt. Beim Öffnen der Datei werden dem Nutzer zwei Ansichten zur Verfügung gestellt. Und zwar die XML Ansicht und die Form Ansicht. In der Form Ansicht werden alle angelegten Elemente in Form einer übersichtlichen Baumstruktur angezeigt. In der XML Datei können innerhalb des *<application>* Tags weitere Elemente hinzugefügt werden.

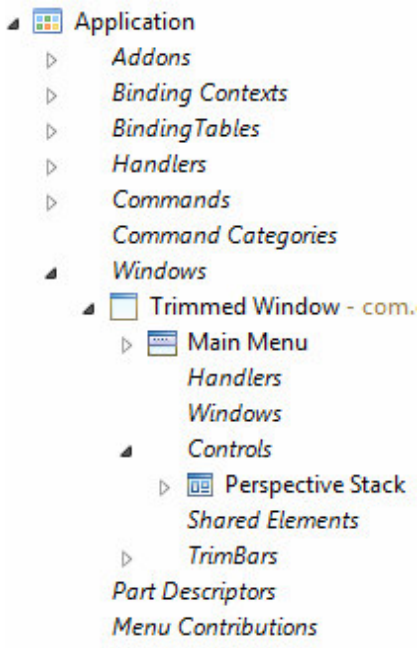


Bild: 5 Form Ansicht des Application Model

Beispiel eines Application Models

Das folgende Beispiel [1] zeigt den Aufbau eines aktiven Application Models mit der dazugehörigen Ausführung. Das Feld *Trimmed Window* im Unterpunkt *Windows* zeigt die aktive View an. Als Kindselement von *Trimmed Window* wird nun der Unterpunkt Main Menu angelegt, der in der Grafik die obere Leiste mit den beiden weiteren Unterpunkten *File* und *Help* bildet. Die Felder *File* und *Help* erhalten beide ein Dropdown-Menu, in denen weitere Unterpunkte definiert werden.
Für die vier Unterpunkte *Open, Save, Quit* und *About* wird zusätzlich jeweils eine zugehörige *Handler-Klasse* im Unterpunkt *Handlers* festgelegt. Diese Handler Klassen werden beim Anklicken des jeweiligen Menüpunktes in der View aufgerufen.

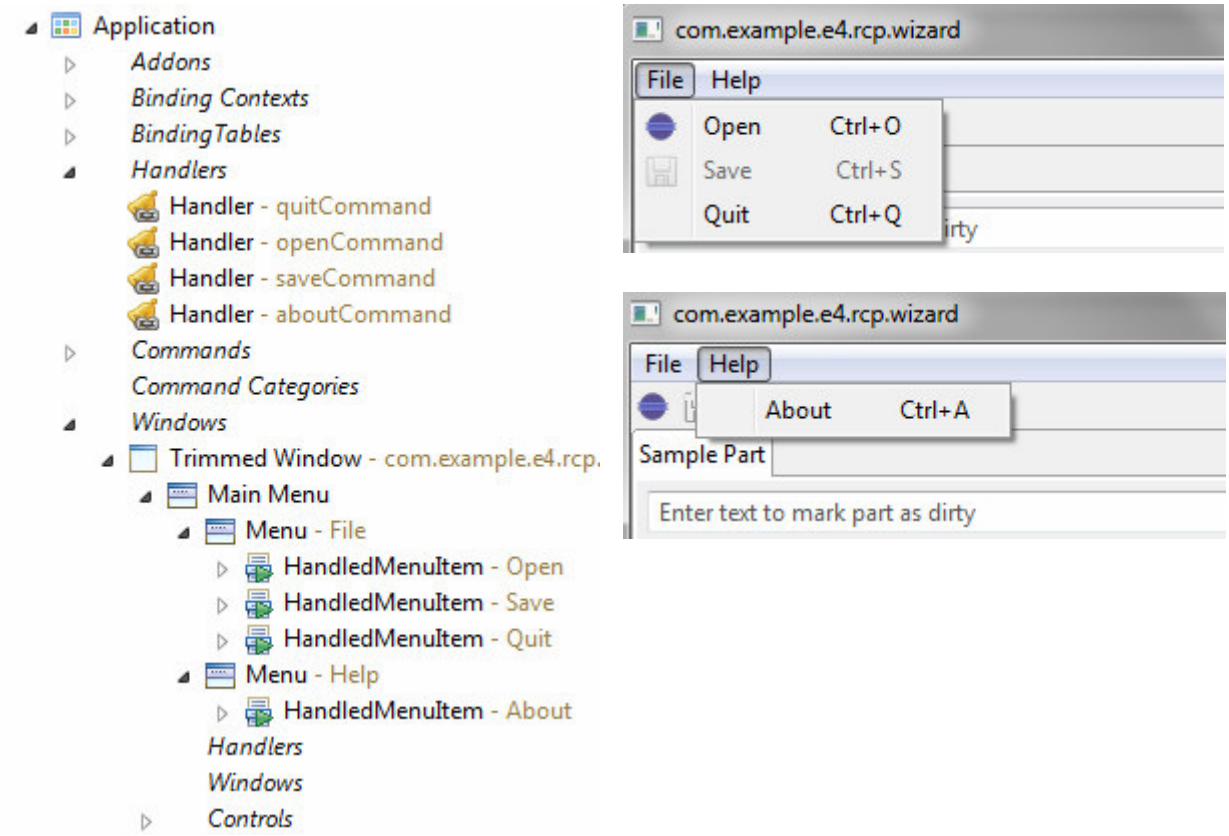


Bild: 6 Beispiel eines Application Models

2.2.3 Unterstützung von CSS Dateien

Eclipse 4 bietet die Möglichkeit das Aussehen von Anwendungen mittels CSS Dateien anzupassen.[11] Bei CSS (Cascading Style Sheets) werden in extern angelegten Dateien sämtliche Eigenschaften zur optischen Gestaltung einer Anwendung eingetragen. Zu diesen Eigenschaften können beispielsweise die Farbe, der Schrifttyp oder die Positionierung von Textfeldern gehören.
CSS hat sich bereits bei der Verwendung von Internetseiten mit HTML bewährt. Gerade die Trennung der Dokumentationen hat sich als sehr leistungsstark erwiesen. Um CSS Dateien innerhalb von Eclipse nutzen zu können, muss in der *plugin.xml* festgelegt werden, dass eine CSS Datei existiert, welche für die optische Darstellung dieser Anwendung zuständig ist. Dies geschieht mit dem folgenden Kommando:

```
platform:/plugin/org.wiedas.fhd.rcpbasis/css/default.css
```

2.2.4 Zukunft von Eclipse [1]

Da es für Eclipse 3 keine weiteren Entwicklungen geben wird, ist es von Entwicklerseite in den Wartungsmodus versetzt worden. Der Rückstand zu den Eclipse 4 Versionen soll aber auch in Zukunft gering gehalten werden. Auch in Zukunft sollen Fehler in Eclipse 3 behoben werden. Support für Eclipse 3 wird es aber von offizieller Seite nicht mehr geben und muss zusätzlich als Long Time Support gebucht werden.
In zukünftigen Eclipse Versionen sollen die Migrationsmöglichkeiten für gemischte Eclipse Programme mit Eclipse 3 und 4 Elementen stärker unterstützt werden und auch ein offizieller Bestandteil von Eclipse werden.

Migration

3.1 Warum migrieren? [8]

Die erste Frage die sich stellt, wenn man beschließt seine Anwendung auf die neue Eclipse Version anzupassen, ist sicherlich warum sollte ich mein Programm migrieren und welche Vorteile habe ich davon. Eclipse 4 bietet wie im oberen Kapitel beschrieben eine Reihe von nützlichen neuen Möglichkeiten zur Programmierung. Gerade das Nutzen von Dependency Injections kann den Programmcode deutlich reduzieren und einfacher gestalten. Die in Eclipse 3 Versionen noch übliche Vorgehensweise eine große Anzahl von verteilten Singletons zu verwenden, kann hierdurch umgangen werden. Ebenso erlauben Dependency Injections einen erleichterten Testlauf der Programme.
Eclipse 4 Anwendungen sind übersichtlicher und strukturierter aufgebaut als Eclipse 3 Anwendungen, was auch an der Einführung des neuen Application Models liegt. Was bei Eclipse 3 noch in einer Mischung aus Code und XML Dateien dargestellt wurde, wird in Eclipse 4 in ein übersichtliches EMF (Eclipse Modeling Framework) eingefügt. Die Möglichkeit für die optische Gestaltung auf CSS Dateien zugreifen zu können ist ein weiterer Grund für die Migration.
Eclipse 3 wird in Zukunft keine Erweiterungen und zusätzlichen Funktionen mehr erhalten[8] und auch der Support der jetzigen Version 3.7 wird vermutlich in den kommenden Jahren eingestellt werden. Hierdurch können Sicherheitslücken oder Kompatibilitätsprobleme mit neuen Anwendungen entstehen die nicht mehr behoben werden.

3.2 Was muss migriert werden? [12]

Wie groß der Aufwand bei der Migration eines Programmcodes ist, hängt stark davon ab, wie gut der Code strukturiert ist. Eine Aufteilung in Bundles für die einzelnen Teile des Programms, wie in der Grafik auf der rechten Seite, erspart eine Menge Aufwand bei der Migration. Hierbei ist gerade die Trennung der Kernkomponenten vom User Interface besonders wichtig. Zusätzliche Funktionen, die nicht durch die Standard RCP abgedeckt sind, sollten klar erkenntlich sein.

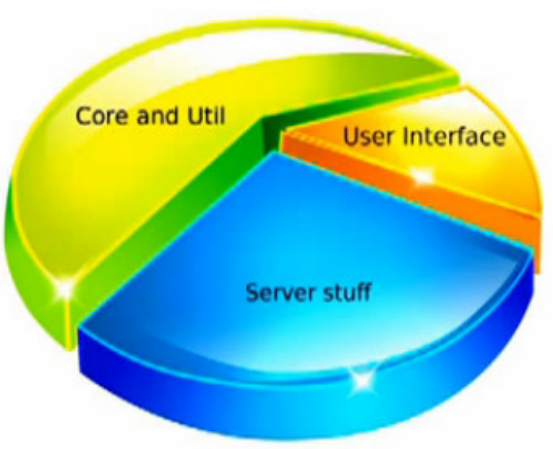


Bild: 7 Programmaufteilung

Bei der Migration nach Eclipse 4 müssen die User Interface Bestandteile des Programms immer migriert werden. Weitere Bestandteile des Programms, wie beispielsweise die Core and Util Klassen, müssen nur dann migriert werden, wenn neue Funktionen von Eclipse 4 verwendet werden sollen. In den meisten Fällen werden die Kernbestandteile nur dann migriert, wenn dort in Zukunft mit Dependency Injections gearbeitet werden soll.

3.3 Migrationsmöglichkeiten von Eclipse 3 nach Eclipse 4 [3,8]

Um seinen Programmcode auf einer Eclipse 4 Plattform lauffähig zu gestalten, stehen dem Nutzer mehrerer Möglichkeiten zur Verfügung. Die verschiedenen Möglichkeiten unterscheiden sich hierbei in der Komplexität der Implementierung, ebenso wie in den Nutzungsmöglichkeiten die nach einem Abschluss der Migration zur Verfügung stehen. Die Bandbreite der Migrationsmöglichkeiten reicht hierbei von der einfachen Nutzung der Kompatibilitätsschicht bis hin zu einer kompletten Übersetzung in Eclipse 4 Code. In diesem Kapitel werden die Migrationsmethoden kurz beschrieben und erklärt welche Möglichkeiten dem Nutzer nach einer Migration mit dieser Methode zur Verfügung stehen. Eine Durchführung der hier gezeigten Migrationsmethoden wird im Folgekapitel durchgeführt.
Bevor eine der gezeigten Methoden auf das eigene Programm angewendet wird, sollte auf jeden Fall ein weiterer Workbench mit einer Kopie des Ursprungsprogramms angelegt werden.

3.3.1 Möglichkeit 1: Nutzung der Kompatibilitätsschicht

Die Möglichkeit der Nutzung der Kompatibilitätsschicht ist die schnellste und einfachste Möglichkeit seinen Code auch in Eclipse 4 nutzen zu können. Viele Projekte für die lediglich eine Lauffähigkeit mit Eclipse 4 sichergestellt werden soll, wird am Anfang diese Methode verwendet werden.
Um dies zu realisieren, gibt es in Eclipse 4 Versionen eine spezielle Schicht die es ermöglicht Programme, deren Code in Eclipse 3 Varianten verfasst wurde, auch in Eclipse 4 Versionen lauffähig zu gestalten. Diese Schicht wird Compatibility Layer (Kompatibilitätsschicht) genannt. Die Kompatibilitätsschicht übersetzt den Programmcode des Eclipse 3 Programms und legt im Hintergrund ein eigenes Application Model mit allen relevanten Informationen an. Für Eclipse selbst entsteht somit der Eindruck, als würde eine Eclipse 4 Anwendung gestartet werden. Beim Nutzen der Kompatibilitätsschicht muss der Programmcode des in Eclipse 3 geschriebenen Programms nicht verändert werden. Ein großer Vorteil der hierdurch entsteht, ist das ein solches Programm auch weiterhin in Eclipse 3 Versionen nutzbar ist.

Allerdings bietet ein Einbinden des Programms über diese Möglichkeit auch keinen Zugang zu den in Eclipse 4 hinzugefügten Neuerungen, wie beispielsweise die Nutzung von Dependency Injections. Die Migration funktioniert nur, wenn ausschließlich Workbench API verwendet werden. Die Möglichkeiten das Aussehen mit CSS Dateien anzupassen existiert auch bei einer Migration allein durch die Kompatibilitätsschicht.

Um sein Programm auf diese Weise einzubinden, werden die folgenden zusätzlichen Plug-Ins benötigt. Diese Plug-Ins müssen eventuell noch manuell in den run-configurations eingestellt werden, da keine direkte Abhängigkeit besteht. [1]

- `org.eclipse.emf.ecore`
- `org.eclipse.emf.common`

Wenn das Programm auf OSGI basiert, muss zusätzlich sichergestellt werden, dass folgende Plug-Ins aktiviert sind.

- `org.eclipse.equinox.ds`
- `org.eclipse.equinox.event`
- `org.eclipse.equinox.util`
- `org.eclipse.e4.ui.workbench.addons.swt`
- `org.eclipse.platform`
- `org.eclipse.ui.forms`
- `org.eclipse.ui.intro` .

Die Durchführung einer Migration mithilfe der Kompatibilitätsschicht wird im Kapitel *"Migration mithilfe der Kompatibilitätsschicht"* beschrieben.

3.3.2 Möglichkeit 2: Nutzung der Kompatibilitätsschicht zusammen mit Eclipse 4 Plug-ins

Bei dieser Methode handelt es sich um eine Mischung aus Eclipse 3 und Eclipse 4 Bestandteilen. Alle Programmteile die in Eclipse 3 Versionen entstanden sind, werden über die Kompatibilitätsschicht für Eclipse 4 zugänglich gemacht. Neue Komponenten können sowohl mit Eclipse 3, als auch mit Eclipse 4 Möglichkeiten entwickelt werden. Ebenso können Teile des alten Programmcodes übersetzt werden, ohne jedoch alle Teile anpassen zu müssen. Hierdurch ist es möglich, die neuen Funktionen von Eclipse 4 wie Dependency Injections in das Programm einzubauen.
Das Mischen von Eclipse 3 und 4 Bestandteilen wird offiziell noch nicht von Eclipse unterstützt. Allerdings gibt es mehrere Möglichkeiten eine solche Mischung von Eclipse 3 und 4 Code dennoch durchzuführen.

Variante 1

Bei der ersten Variante werden dem Application Model, welches durch die Kompatibilitätsschicht im Hintergrund erzeugt wurde, zusätzliche Prozesse und fragments hinzugefügt. Allerdings können bei dieser Methode noch zeitliche Probleme auftreten, wenn die Prozesse und die fragments ausgeführt werden, obwohl das Application Model im Hintergrund noch nicht vollständig erstellt worden ist. Diese Variante kann bei handlern und bei Views verwendet werden. Sie funktioniert allerdings nicht bei Editoren.

Variante 2

In der zweiten Variante wird eine Kopie des Application Model angelegt, welches durch die Kompatibilitätsschicht erzeugt wird. Diese Kopie wird als Application Model der Anwendung registriert und es werden neue Eclipse 4 Komponenten hinzugefügt. Das relevante Modell `LegacyIDE.xmi` kann im Plug-in `org.eclipse.ui.workbench.` gefunden werden.

Variante 3

Die dritte Variante funktioniert über eine Brücke zwischen Eclipse 3 und Eclipse 4 Versionen die von Tom Schindl [6] entwickelt wurde. Das Ziel dieser Brücke besteht darin, dass Views und Editoren sowohl in Eclipse 3, als auch in Eclipse 4 parallel genutzt werden können. Um diese Möglichkeit zu aktivieren, muss das Bundle `org.eclipse.e4.tools.compat` zu den aktiven Plug-Ins hinzugefügt werden und die `org.eclipse.e4.tools.e3x.bridge` muss installiert werden.
Bei der Migration wird jede View innerhalb des Programms auf Eclipse 4 angepasst. Hierbei können sowohl Eclipse 3 als auch Eclipse 4 Codebausteine verwendet werden. Für jede auf diese Weise angepasste View wird eine Wrapper Klasse zur Übersetzung des Eclipse 3 Codes angelegt. Die Wrapper Klasse wird anschließend anstelle der View als Extension Point eingesetzt.

Ein Beispiel für das Verwenden dieser Variante wird auch bei der Migration der Nutzerwelten RCP verwendet und ausführlich im Kapitel *"Migration mithilfe der e3x Toolbridge"* beschrieben.

3.3.3 Möglichkeit 3: Eine reine Eclipse 4 Applikation [9]

Bei der dritten Form der Migration werden alle Bestandteile, die ausschließlich in Eclipse 3 Versionen verwendet werden, umgeschrieben in Eclipse 4 Bestandteile. Das Ziel bei dieser Form der Migration besteht darin, dass nach Abschluss der Umformung die Kompatibilitätsschicht nicht mehr genutzt werden muss. Nach Abschluss dieser Migration, können alle Funktionen von Eclipse 4 verwendet werden, da es sich jetzt um einen reinen Eclipse 4 Code handelt. Auf der anderen Seite bedeutet eine solche Umwandlung auch das Komponenten die nur in Eclipse 3 vorhanden waren nicht weiter eingesetzt werden können. Beispiele für diese Komponenten sind das Error Log und die Konsolen View.
Am einfachsten lässt sich diese Methode für neu angelegte Projekte verwenden da hierfür vom Start an Eclipse 4 Code verwendet wird.

Um ein bestehendes Programm mit dieser Methode zu migrieren, müssen einige Schritte durchgeführt werden. Zunächst müssen wir für jedes unserer zu migrierenden Bundles eine `Application.e4xmi` Datei hinzufügen. Hiermit wird in den jeweiligen Plug-Ins ein Application Model erzeugt. Anschließend müssen alle Bestandteile des Codes Schritt für Schritt je nach ihrem Typ migriert werden (control , view ...). Bei der Anpassung des Codes gibt es einige wichtige Punkte die erfüllt werden müssen: [9]

- Alle Singletons innerhalb des Programms müssen entfernt werden und durch Injektionen ersetzt werden.
- Alle Services müssen injiziert werden.
- Alle Eclipse 3 Auswahlmechanismen müssen durch Eclipse 4 Varianten ersetzt werden.

Ein Beispiel für die Durchführung einer Migration mit dieser Möglichkeit wird im Kapitel *" Eine reine Eclipse 4 Migration "* durchgeführt.

Migration der Nutzerwelten RCP

4.1 Welche Migration wird durchgeführt

Zunächst muss entschieden werden, welche Art der Migration durchgeführt werden soll. Die Entscheidung welche Migrationsmethode am besten zu dem bestehenden Programm passt, muss abhängig vom jeweiligen Projekt entschieden werden. Wichtige Faktoren sind hierbei ob ein Projekt schon viele Komponenten besitzt, die migriert werden müssen, oder ob es sich um ein neues Projekt handelt, dass mit nur wenig Aufwand für Eclipse 4 umgeschrieben werden kann. Ein wichtiger Faktor ist auch die geplante Laufzeit des Projekts um zu entscheiden, ob sich eine Migration des Programms lohnt.
In den folgenden Kapitel werden die benötigten Schritte beschrieben die für eine Migration mithilfe der Kompatibilitätsschicht und mit der *e3x Toolbridge* benötigt werden. Anschließend werden die Schritte die beim Durchführen einer reinen Eclipse 4 Migration benötigt werden aufgezeigt. Unabhängig von der gewählten Migrationsmethode muss zunächst ein neuer Workbench angelegt werden.

4.2 Anlegen des Workbench

Der erste Schritt um mit der Migration zu starten, ist das Anlegen eines neuen Workbenches für Eclipse 4. Hierfür muss zunächst eine neue Version der Eclipse RCP heruntergeladen und separat von der alten Version abgespeichert werden. Die aktuelle Eclipse Version kann auf der Website *https://eclipse.org* gefunden werden. Als nächstes wird eine Kopie des Eclipse 3 Workbenches angelegt und für die Nutzung mit Eclipse 4 umbenannt. Es sollte auf jeden Fall für die Migration in einer Kopie des Originalprogramms gearbeitet werden, da es beim Programmieren zu Fehlern kommen kann und das Programm nach Abschluss der Migration nicht mehr in Eclipse 3 nutzbar ist.
Wenn dem Workbench zusätzliche Plug-Ins hinzugefügt wurden, müssen diese auch in einer neuen Eclipse Version erneut hinzugefügt werden. Hierbei muss vorher sichergestellt werden, ob diese neuen Plug-Ins auch mit der neuen Eclipse Version kompatibel sind, oder ob in der neuen Eclipse Version schon ein Äquivalent diese Plug-Ins existiert.

4.3 Migration mithilfe der Kompatibilitätsschicht

Die Migration mithilfe der Kompatibilitätsschicht ist die einfachste Variante seinen Programmcode zu migrieren. Diese Variante ist vor allem dann nützlich, wenn zwar als Umwicklungsumgebung mit Eclipse 4 gearbeitet werden soll, aber keine Elemente von Eclipse 4 benötigt werden. Des Weiteren wird die Nutzung der Kompatibilitätsschicht benötigt, um eine Migration mit der e3x Toolbridge durchzuführen.
Eine Migration der kompletten Nutzerwelten RCP mit dieser Methode, ohne eine weitere Nutzung durch die Toolbridge, würde derzeitig keinen großen Nutzen für das Projekt bringen. Alle Komponenten des Projektes sind auf Eclipse 3 ausgelegt. Eine grafische Anpassung der Views mittels CSS Dateien ist in nächster Zeit ebenfalls nicht vorgesehen.

Schritte zur Migration mit der e3x Toolbridge

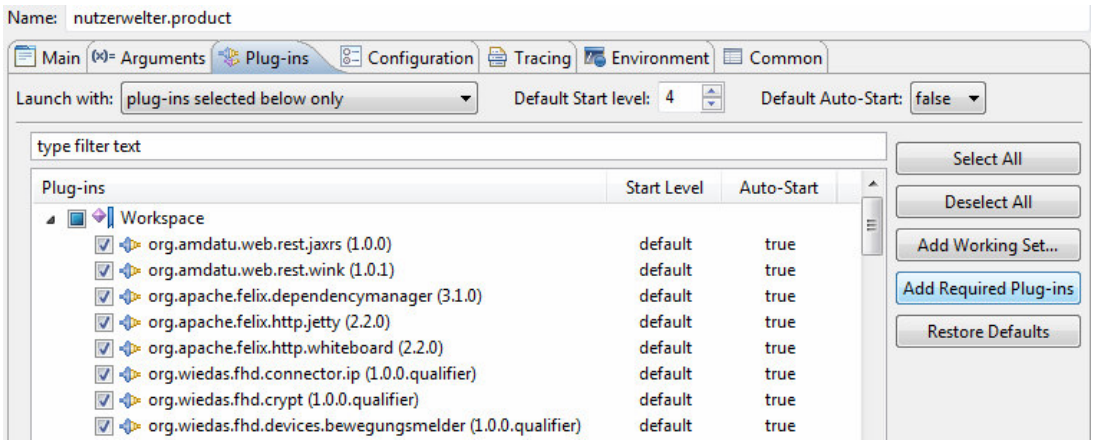
- **Schritt 1:** Hinzufügen fehlender Plug-Ins
- **Schritt 2:** Versionsnummern anpassen

4.3.1 Schritt 1: Hinzufügen fehlender Plug-Ins

Der erste Schritt für die Migration der Nutzwelten RCP ist die Migration mithilfe der in Eclipse 4 vorhandenen Kompatibilitätsschicht. Hierbei wird der Code der Nutzerwelten RCP wie im Unterpunkt *"Nutzung der Kompatibilitätsschicht"* beschrieben nicht verändert.

Um den Code lauffähig zu gestalten, müssen ein paar Änderungen an den vorhandenen Plug-Ins vorgenommen werden und noch benötigte Plug-Ins hinzugefügt werden.

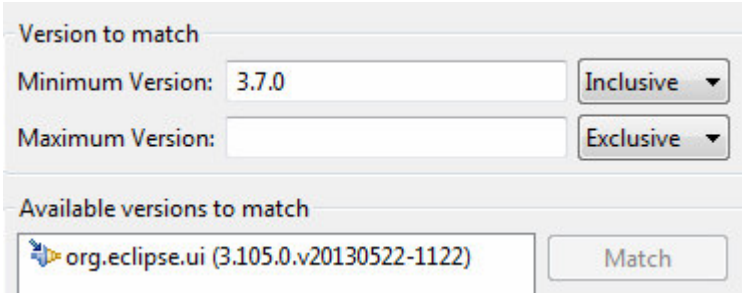
Da unser Code auf OSGI basiert, müssen alle im Unterpunkt *"Nutzung der Kompatibilitätsschicht"* erwähnten Plug-Ins in unsere run configuration eingebunden werden. Um noch fehlende Plug-Ins hinzuzufügen, wählen wir in den run configurations den Unterpunkt *"Add required Plug-ins"*. Hiermit werden alle noch benötigten Plug-Ins aktiviert soweit diese vorhanden sind.



4.3.2 Schritt 2: Versionsnummern anpassen

Bei einem Start des Programms kann es vorkommen, dass manche der Plug-Ins, die von Eclipse 4 zur Verfügung gestellt werden, nicht von den Bundles im Programm gefunden werden.
Um sicherzustellen das die Nutzerwelten RCP sowohl in 3.7 also auch in höheren Eclipse Versionen funktioniert, muss sichergestellt werden, dass alle Bundles auch die höheren Versionen von Eclipse 4 unterstützen. Hierfür müssen alle Manifest Dateien der Bundles innerhalb des Programms aufgerufen werden und der Unterpunkt Dependencies angewählt werden. Anschließend wird die Version 3.7 als Minimum Version eingetragen und somit auch die Möglichkeit gewährt alle höheren Versionen Nutzen zu können.

In diesem Beispiel sehen wir das Eclipse nur die Version 3.105.0 zur Verfügung stellt. Da wir die Version 3.7 als Minimum gesetzt haben, kann diese aber auch verwendet werden.



Nach Durchführung der oben genannten Schritte lässt sich die Nutzerwelten RCP in Eclipse 4 ohne Fehlermeldungen ausführen.

4.4 Migration mithilfe der e3x Toolbridge

Bei der Migration mit der e3x Toolbridge handelt es sich um eine kombinierte Migrationsmethode, um sowohl Eclipse 3, als auch Eclipse 4 Programmteile weiterhin nutzen zu können. Dieses Kapitel beschreibt als erstes die benötigten Schritte für eine Migration und beschreibt diese Schritte anschließend anhand einer Migration des Nutzerwelten RCP Bundles.
Bei dieser Methode können vorhandene Projektteile weiterhin über die Kompatibilitätsschicht ohne eine Anpassung des Codes genutzt werden. Hierdurch eignet sich diese Migrationsmethode gut für das Nutzerwelten Projekt. Die große Anzahl bestehender Bundles für die einzelnen Geräte, sowie das Anlegen der View, mittels Workbench Klassen kann weiterhin genutzt werden.

Schritte zur Migration mit der e3x Toolbridge

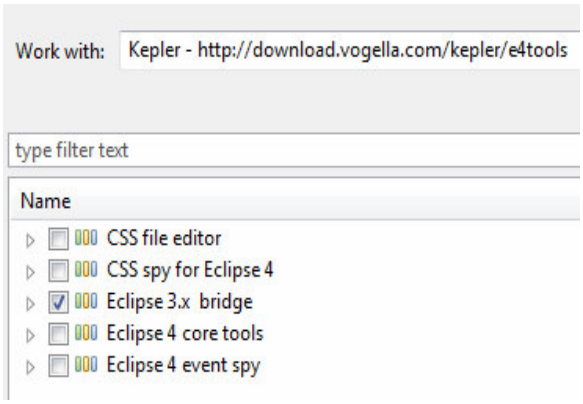
Um eine Migration mithilfe der e3x Toolbridge durchzuführen, müssen die folgenden Schritte durchgeführt werden. Bevor die Toolbridge zur Nutzung von Eclipse 4 Elementen hinzugefügt wird, muss das Programm zunächst mithilfe der Kompatibilitätsschicht lauffähig gestaltet werden. Hierfür müssen die beiden Schritte aus dem Kapitel *"Migration mithilfe der Kompatibilitätsschicht*" durchgeführt werden.

- **Schritt 1:** Installieren der Toolbridge
- **Schritt 2:** Anpassen der Views
- **Schritt 3:** Wrapper Klassen erstellen
- **Schritt 4:** Wrapper Klassen einbinden

4.4.1 **Schritt 1:** Installieren der Toolbridge

Damit zukünftige Projekte der Nutzerwelten RCP in Eclipse 4 Code programmiert werden können, und die Views der Nutzerwelten RCP auch in Eclipse 4 ordnungsgemäß dargestellt werden können, müssen die Views mithilfe einer Brücke in Eclipse 4 migriert werden. [6]

Hierfür muss das Eclipse 4 Plug-In `org.eclipse.e4.tools.compat` zu den aktiven Plug-Ins in der run configuration hinzugefügt werden. Die zur Durchführung benötigte Brücke `org.eclipse.e4.tools.e3x.bridge` kann über den Button "*Help / Install New Software*" von der Eclipse tools update Seite heruntergeladen werden.



(für Eclipse Kepler: Kepler - http://download.vogella.com/kepler/e4tools)

Jedes Bundle in dem eine View dargestellt werden soll, benötigt die folgenden beiden Plug-Ins. Um diese beiden Plug-Ins hinzuzufügen, werden diese in der Manifest Datei des jeweiligen Bundles in den Abschnitt "Required Bundles" eingefügt.

- `org.eclipse.e4.tools.compat`
- `org.eclipse.e4.core.contexts`

4.4.2 Schritt 2: Anpassen der Views

Nach Abschluss dieser Schritte kann eine View als POJO (Plain old Java Object) angelegt werden. Hierbei können auch Funktionen von Eclipse 4 genutzt werden. Um dies zu realisieren wird die Abhängigkeit vom ViewPart entfernt und alle Abhängigkeiten werden mithilfe von Dependency Injections aufgelöst. Das Zuweisen der View zum aktiven Part wird in Zukunft von der Wrapper Klasse übernommen und ist nicht länger Teil des POJO.
Da die Klasse anschließend mit einer Wrapper Klasse von Eclipse 3 in Eclipse 4 Code übersetzt wird, ist es nicht mehr möglich, dass bei der Programmierung des POJO Befehle genutzt werden, die ausschließlich in Eclipse 3 existieren. Des Weiteren müssen alle Forderungen erfüllt werden, die bei einer reinen Eclipse 4 Anwendung gestellt werden. Ein gutes Beispiel hierfür ist die Entfernung aller Singletons.
Hier sehen wir den Programmkopf, wenn er auf die Version 4 angepasst wurde. Um die PartControl mittels Dependency Injections zu erstellen, wird das Schlüsselwort PostConstruct anstelle von Override verwendet.

Der gesamte Programmcode der angepassten View befindet sich im Anhang.

LogoView in Eclipse 3.7

```
public class LogoView extends ViewPart {

    public static final String ID = "org.wiedas.fhd.rcpbasis.views.LogoView"; //$NON-NLS-1$

    public LogoView() {
    }

    @Override
    public void createPartControl(Composite parent) {
```

LogoView in Eclipse 4.3

```
public class LogoView {

    public static final String ID = "org.wiedas.fhd.rcpbasis.views.LogoView"; //$NON-NLS-1$

    public LogoView() {
    }

    @PostConstruct
    public void createPartControl(Composite parent) {
```

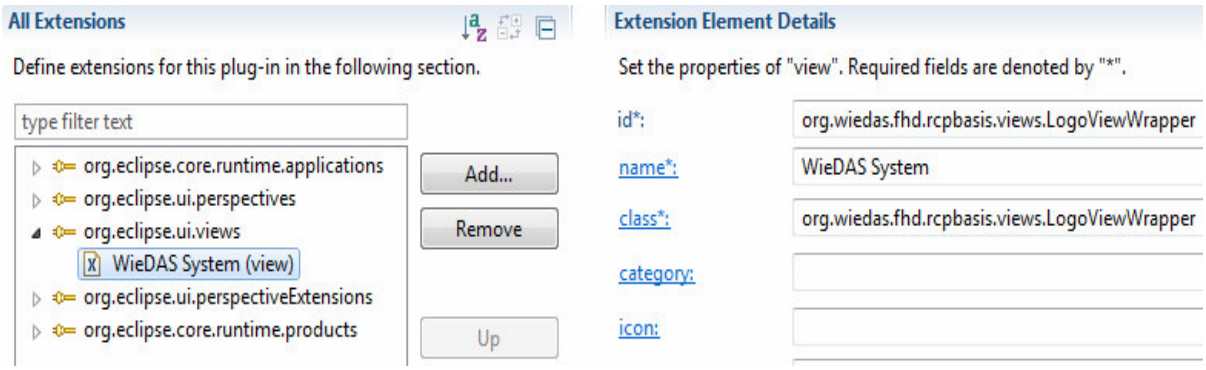
4.4.3 Schritt 3: Wrapper Klassen erstellen

Im Anschluss an die Anpassung der View wird eine weitere Klasse erstellt mit dem Namen "ViewWrapper". Diese Klasse enthält den Code um den Inhalt der in Eclipse 4 verfassten View zu importieren und anschließend in eine für Eclipse 3 lesbare Form zu übersetzen.
In Eclipse 4 werden die Views über das Application Model automatisch den passenden Parts zugeordnet. Da unsere Views weiterhin in einem Eclipse 3 Interface angezeigt werden sollen, ist es nötig, dass die Wrapper Klasse das Eclipse 4 POJO an der passenden Stelle als ViewPart einfügt.
Für jede Klasse, in der die neuen Funktionen von Eclipse 4 genutzt werden sollen, muss eine zuständige Wrapper Klasse angelegt und eingebunden werden. Die Wrapper Klasse muss sich hierbei im selben Bundle befinden wie die Klasse die übersetzt werden soll. Das folgende Beispiel zeigt den Inhalt der Wrapper Klasse der rcpbasis view.

```
package org.wiedas.fhd.rcpbasis.views;

import org.eclipse.e4.tools.compat.parts.DIViewPart;
import org.wiedas.fhd.rcpbasis.views.LogoView;;


public class LogoViewWrapper extends DIViewPart {
    public LogoViewWrapper() {
         super(LogoView.class);
        }

}
```

4.4.4 Schritt 4: Wrapper Klassen einbinden

Die Klasse ViewWrapper wird nun anstelle der Original View als Extension in die plugin.xml Datei geschrieben. Um dies durchzuführen wird die plugin.xml Datei des RCP Bundles ausgewählt. Im Unterpunkt Extensions werden im Verweis des Bundles "*org.eclipse.ui.views*" der id und der class Eintrag durch den der Wrapper Klasse ersetzt.



Durch das Einsetzen der Wrapper Klasse als Extension Point können nun alle anderen Bestandteile des Programms den Code aus der View so benutzen, als handle es sich hierbei um Eclipse 3 Code. Somit ist dieser auch in einem 3.x Interface nutzbar.

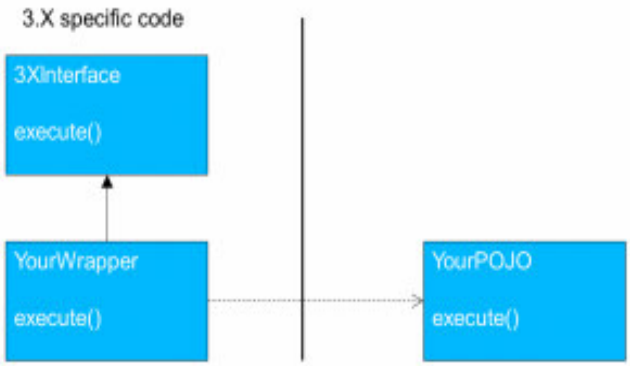


Bild: 8 View Wrapper

4.5 Eine reine Eclipse 4 Migration

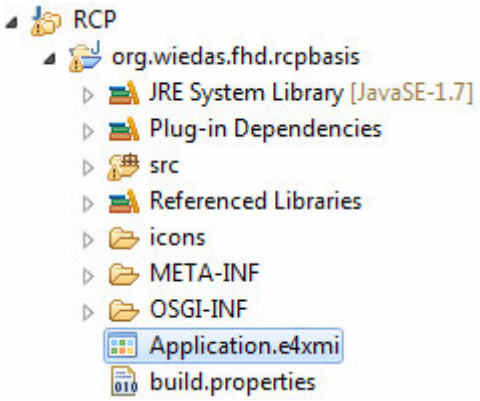
Bei einer reinen Eclipse 4 Migration werden alle Bestandteile des Programmcodes auf Eclipse 4 angepasst. Nach Abschluss der Migration soll die Kompatibilitätsschicht nicht mehr benötigt werden.
Für bereits bestehende Projekte kann der Aufwand der Programmanpassung je nach Komplexität des Programms ziemlich groß werden. Das Anpassen des Codes ist von Programm zu Programm verschieden und kann sogar dazu führen, dass Teile des Programms umgeschrieben werden müssen, wenn es keine Alternativen für Programmcodekomponenten von Eclipse 3 gibt. Für das Nutzerwelten Projekt eignet sich diese Migrationsmethode daher weniger gut. Durch die Aufteilung des Nutzerwelten Projekts in einzelne Bundles, kann es darüber hinaus nötig sein, das Application Model in einzelne Fragmente zu den jeweiligen Bundles zu unterteilen.
Für das aufgeführte Beispiel wird lediglich das RCP Bundle migriert. Daher wird in dieser Variante auch nur ein Application Model benötigt und kein in Fragmente aufgeteiltes Modell.

Schritte für eine reine Eclipse 4 Migration

- **Schritt 1:** Anlegen eines Application Models
- **Schritt 2:** Implementierung des Application Models
- **Schritt 3:** Anpassen des Codes

4.5.1 Schritt 1: Anlegen eines Application Models

Bei der Migration der Nutzerwelten RCP als reine Eclipse 4 Anwendung können die Komponenten der Benutzeroberfläche wie beispielsweise die views und die editoren unserer `plugin.xml` Datei nicht einfach ohne Änderungen wiederverwendet werden. Stattdessen müssen diese Komponenten über die Extension Points in das Application Model eingefügt werden. Anschließend müssen alle Singletons, innerhalb der betroffenen Klassen, durch Dependency Injections ersetzt werden.



Um unser Application Model für das RCP Bundle unserer Nutzerwelten RCP anzulegen, gehen wir mit einem Rechtsklick auf das `org.wiedas.fhd.rcpbasis` Bundle und wählen `New/other/Eclipse4/Model/New Application Model` aus. Der Name des Application Models bleibt standardmäßig bei `Application.e4xmi`.
Damit im angelegten Application Model auch zusätzliche Funktionen wie das Command und Handler Framework sowie die Drag-and-Drop Funktion zur Verfügung stehen, müssen im Application Model einige Addon-Klassen registriert sein. Damit diese Addons standardmäßig genutzt werden können, empfiehlt es sich die Checkbox *Include default addons* immer zu aktivieren.

4.5.2 Schritt 2: Implementierung des Application Models

Der nächste Schritt ist es unser Application Model zu füllen und alle Verbindungen einzufügen. Die Implementierung des Application Models kann je nach Komplexität des Programms einen großen Aufwand bedeuten. Um diesen Vorgang zu erleichtern, kann alternativ zum manuellen Erstellen der Verbindungen das Application Model auch über die Kompatibilitätsschicht und einen live model editor beschrieben werden.

Variante 1: Verbindungen manuell erstellen

Die erste Variante der Implementierung des Application Models ist die schrittweise Anpassung durch manuelles Erstellen der Verbindungen. Bei dieser Variante wird der XML Code direkt in das Application Model eingetragen. Um nicht den gesamten XML Code selbst schreiben zu müssen, steht hierfür das Tool "Application Model Editor" zur Verfügung.

Bei Eclipse 4 wird die View über das Fenster Windows im Application Model realisiert und die Workbench Klassen, die für die Darstellung in Eclipse 3 notwendig waren, werden nicht länger benötigt. Alle Einstellungen von Größen und Positionen der jeweiligen Fenster können im Application Model festgelegt werden.
Bei der Erstellung des Application Model mit dem "*Application Model Editor*" wird standardmäßig eine aktive View mit dem Namen "*Trimmed Window*" erstellt. Um die Ansichten für die einzelnen Views zu erstellen, müssen diese nun mit dem Erstellen von "*Parts*" als Kindselemente der aktiven View angelegt werden. Beim Hinzufügen weiterer Elemente durch diese Methode, wird von Application Model automatisch der benötigte XML-Code generiert.
Um eine View oder einen handler aus der alten Eclipse 3.x Version in Eclipse 4 zu übernehmen, muss der entsprechende Part innerhalb der View des Application Models mit Rechtsklick angewählt werden. Anschließend wählt man die Option "*Import 3.x*" und kann dort die Unterpunkte View oder handler auswählen. Wenn man hier die entsprechende Option auswählt, erscheint eine Liste der verfügbaren Plug-Ins.

Variante 2: Nutzung des live model editors

Wenn ein Eclipse 3 Programm mithilfe der Kompatibilitätsschicht in Eclipse 4 übersetzt wird, legt die Kompatibilitätsschicht bei der Übersetzung eine eigene Version für das Application Model an, die nur intern für die Übersetzung benutzt wird. Bei der Nutzung des live model editors wird diese Tatsache ausgenutzt, um die Implementierung des Application Models zu erleichtern.[9] Die Übersetzung der Kompatibilitätsschicht wird abgefangen und im eigenen Projekt verwendet.

Zusätzlich zu dem Workbench für die reine Eclipse Migration wird ein zweiter Workbench angelegt. In diesem Workbench müssen jetzt die nötigen Schritte durchgeführt werden, damit die Anwendung mithilfe der Kompatibilitätsschicht läuft.
Um die Kompatibilitätsschicht nutzen zu können, müssen alle im Kapitel *"Nutzung der Kompatibilitätsschicht"* erwähnten Plug-Ins in den run-configurations hinzugefügt werden. Um den live model editor zu benutzen, wird noch zusätzlich das Plug-In *`"org.eclipse.e4.tools.emf.liveeditor"` geladen.*

Um das Application Model zu implementieren, müssen nun die folgenden Schritte durchgeführt werden:[9]

- Starten der Eclipse 3 Anwendung.
- Den Live Editor öffnen durch drücken von Alt+Shift+F9.
  (Hinweis: Hierfür muss die Verwendung von Tastenkürzeln aktiviert sein)
- Den XML Code für die Komponenten in der Eclipse 3 Anwendung herausfiltern und kopieren.
- Die Bestandteile des XML Codes in das Application Model der reine Eclipse 4 Migration einfügen.
- Anpassen der XML Datei (beispielsweise müssen für Views die Links zu den entsprechenden POJOS angegeben werden).

Beim Anpassen der XML Datei müssen auch die Geräte innerhalb der View entsprechend den Bedürfnissen angepasst werden.

4.5.3 Schritt 3: Anpassen des Codes

Im Anschluss an die Implementierung muss das Application Model noch in der plugin.xml Datei als zugehöriges Model definiert werden. Hierfür muss beim Unterpunkt "*application*" der XML Datei das neue Modell eingetragen werden. Ein Beispiel hierzu befindet sich im Unterkapitel "*Das Application Model*".

Das Application Model sollte nun vollständig eingebunden sein und die Kompatibilitätsschicht in der Anwendung nicht aktiviert sein. Wenn dies der Fall ist, werden in den Klassen, die für die Darstellung der Views in Eclipse 4 benötigt werden, eine Reihe von Fehlermeldungen angezeigt. Diese Fehlermeldungen gilt es nun durch das Einsetzen von Eclipse 4 Code zu beseitigen. Für das Einbinden von Services, muss der Code beispielsweise durch Dependency Injections ersetzt werden. Das Setzen eines Fokus für die aktive View ist in Eclipse 4 nicht mehr nötig und kann daher gelöscht werden. An vielen Stellen können zusätzlich zur Fehlerkorrektur Teile des Programmcodes die nicht mehr benötigt werden weggelassen werden.
Komplizierter wird die Darstellung, wenn es sich um Fehlermeldungen handelt, bei denen Techniken benutzt wurden die es in Eclipse 4 nicht mehr gibt. In einem solchen Fall müssen Eclipse 4 Äquivalente zu den alten Kommandos gefunden werden.

Fazit:

Durch eine Migration des bestehenden Programms in Eclipse 4 stehen dem Nutzer eine Reihe von nützlichen neuen Funktionen zur Verfügung. Diese neuen Funktionen vereinfachen das Programmieren und die Struktur des Programmcodes. Ob sich eine Migration zur neuen Eclipse Version lohnt, ist jedoch von Projekt zu Projekt unterschiedlich. Entscheidend dafür, ob eine Migration durchgeführt wird, ist vor allem die Laufzeit des Projektes, sowie die Komplexität des Aufbaus. Eclipse 4 bietet hierfür eine Reihe unterschiedlicher Möglichkeiten zum Migrieren zur Verfügung, wobei sich der Aufwand der einzelnen Methoden voneinander unterscheidet.

Wie lange das Nutzerwelten Projekt noch in Betrieb sein wird, steht noch nicht fest. Des Weiteren ist der Programmcode des Projekt in viele unterschiedliche Bundles für die jeweiligen Geräte aufgeteilt. Falls sich für eine Migration der Nutzerwelten RCP entschieden werden sollte, wäre eine gemischte Migration, die sowohl Eclipse 3 als auch Eclipse 4 Bestandteile hat, eine gute Lösung. Der Programmieraufwand wäre deutlich geringer, als bei einer reinen Eclipse 4 Umsetzung, da alle alten Bestandteile bestehen bleiben könnten. Neue Programmteile könnten dennoch mit Eclipse 4 Methoden geschrieben und eingebunden werden. Ein weiterer Vorteil dieser Migrationsmethode besteht darin, das auch die Workbench Klassen zur Erzeugung der View in ihrer jetzigen Form erhalten bleiben können und nicht auf ein Application Model Fenster in Eclipse 4 geändert werden müssen. Neu erstellte Views würden somit, wie im Beispiel erläutert, mithilfe einer Wrapper Klasse für den Eclipse 3 Code übersetzt werden

Quellenverzeichnis:

Als Quellen für die vorliegende Arbeit wurden als Hauptnachschlagewerke die beiden Eclipse Bücher *"Rich Clients mit dem SDK 4.2"* sowie *"Eclipse 4 RCP"* verwendet. Zusätzlich wurden Informationen aus Internettutorien und Artikeln gesammelt.

[1] **Eclipse 4 RCP** The complete guide to Eclipse application development Second Edition based on Eclipse 4.3

Vogella Series Lars Vogel ISBN: 9783943747072

[2] **Eclipse 4** Rich Clients mit dem Eclipse SDK 4.2

Entwickler.press Marc Teufel und Dr. Jonas Helming ISBN: 978-3-86802-063-2

[3] **Eclipsesource**

(Eclipse 4 Tutorials , Migrationsmethoden)

http://developer.eclipsesource.com/
http://eclipsesource.com/blogs/

[4] **Dirks Metric** Eclipse RCP e4 with 3.x views

(Eclipse 4 Migration mit Eclipse 3 Elementen)

http://dirksmetric.wordpress.com/2012/08/01/tutorial-eclipse-rcp-e4-with-3-x-views-like-project-explorer-properties-etc/

[5] **Ralf Ebert** Entwicklung von Desktop Anwendungen mit der Eclipse Rich Client Platform 3.7

(Eclipse IDE / Eclipse RCP , Vergleiche von Eclipse 3 zu Eclipse 4)

http://www.ralfebert.de/archive/eclipse_rcp/EclipseRCP.pdf

[6] **Tom Schindl** How to Apply the e4 Programming model to 3.x

(Erklärung und Installation der Eclipse 3.x Toolbridge)

http://tomsondev.bestsolution.at/2011/06/10/how-to-apply-the-e4-programming-model-to-3-x/

[7] **Nutzerwelten Projekt Team** Nutzerwelten Kick Off Präsentation
(WiesDAS / Nutzerwelten Projekt)
NutzerWelten_Kick-Off-Präsentation Intro_Stand 20140307

[8] **Lars Vogel** Migrationsguides
(Migrationsguides , Erklärungen zu Migrationsmöglichkeiten)
http://www.vogella.com

[9] **Eclipsecon**
(Migartionserklärungen der Eclipsecon 2013 und 2014)
http://www.eclipsecon.org/

[10] **Eclipse** Offizielle Eclipse Homepage
(Anpassung der View in Eclipse 4 , Fehlersuche , Download der Software)
http://www.eclipse.org

[11] **Eclipse Wiki**
(Eclipse CSS und RCP , Nutzung der Kompatibilitätsschicht)
http://wiki.eclipse.org

[12] **Eclipse Foundation** Migration to e4 - be aware of the pitfalls
(Fehler bei der Migration , Erklärung einer einen Eclipse 4 Migration)
https://www.youtube.com/watch?v=RmBj3N7aNN4

[13] **Oracle Learning Library** Building Rich Client Application with Eclipse 4
(Aufbau der Eclipse RCP)
https://www.youtube.com/watch?v=IbMVDxgLYdk

[14] **Jakob Jekov** Dependency Injection
(Dependency Injections)
http://tutorials.jenkov.com/dependency-injection/index.html

Bilder :

[Bild 1] OSGI-Bundle Struktur des Informatik Labors

Bachelor Thesis Daniel Andreas 07.08.2013

[Bild 2] Aufbau der Eclipse RCP (Version 3.7)

http://www.ralfebert.de/archive/eclipse_rcp/EclipseRCP.pdf

[Bild 3] Dependency Injections

http://satoricode.net/content/images/DependencyInjection/SimpleDependencyInjection.png

[Bild 4] Aufbau des Eclipse Context

http://eclipsesource.com/blogs/tutorials/eclipse-4-e4-tutorial-part-4-dependency-injection-basics/

[Bild 5] Form Ansicht des Application Model

Eclipse 4 RCP Vogela series von Lars Vogel ISBN:9783943747072

[Bild 6] Beispiel eines Application Models

Eclipse 4 RCP Vogela series von Lars Vogel ISBN:9783943747072

[Bild 7] Programmaufteilung

https://www.youtube.com/watch?v=RmBj3N7aNN4 *(Migration to e4 - be aware of the pitfalls)*

[Bild 8] View Wrapper

http://eclipsesource.com/blogs/tutorials/eclipse-4-e4-tutorial-soft-migration-from-3-x-to-eclipse-4-e4/

Anhang:

LogoView in Eclipse 4.3

```
public class LogoView {

	public static final String ID = "org.wiedas.fhd.rcpbasis.views.LogoView"; //$NON-NLS-1$
//public static OptionsWieDAS owOptions= new OptionsWieDAS(); //8YE

	public LogoView() {
	}

	@PostConstruct
	public void createPartControl(Composite parent) {


		Composite container = new Composite(parent, SWT.NONE);
		container.setLayout(new GridLayout(1, false));

		Label logo = new Label(container, SWT.CENTER);
		logo.addMouseListener(new MouseAdapter() {
			@Override
			public void mouseDoubleClick(MouseEvent e) {
				OptionsWieDAS.OptionsDlg(); //8YE


			}

});
		logo.setImage(ResourceManager.getPluginImage("org.wiedas.fhd.rcpbasis",
		"icons/nutzerwelten.png"));
		GridData gd_logo = new GridData(SWT.FILL, SWT.CENTER, true, false, 0, 1);
		gd_logo.widthHint = 433;
		logo.setLayoutData(gd_logo);

		Label lblVersion = new Label(container, SWT.NONE);
		lblVersion.setLayoutData(new GridData(SWT.CENTER, SWT.CENTER, false, false, 1, 1));
		lblVersion.setText("Nutzer Welten - Studie Sicherheit");

		Label lblNewLabel = new Label(container, SWT.NONE);
		lblNewLabel.setLayoutData(new GridData(SWT.CENTER, SWT.CENTER, false, false, 1, 1));
		lblNewLabel.setAlignment(SWT.CENTER);
		lblNewLabel.setText("Version 0.1 Beta");

		new Label(container, SWT.NONE);

		Label lblBeiProblemenWenden = new Label(container, SWT.CENTER);
		lblBeiProblemenWenden.setLayoutData(new GridData(SWT.CENTER, SWT.CENTER, false, false,
1, 1));
		lblBeiProblemenWenden.setAlignment(SWT.CENTER);
		lblBeiProblemenWenden.setText("Bei Problemen wenden Sie sich an");
```

```
		Label lblFachhochchule = new Label(container, SWT.CENTER);
		lblFachhochchule.addMouseListener(new MouseAdapter() {
		//	@Override
		public void mouseDoubleClick(MouseEvent e) {
				try {


	PlatformUI.getWorkbench().getWorkbenchWindows()[0].getActivePage().showView("org.wiedas.fhd.vie
                                                                                ws.debug.Debug");
				} catch (PartInitException e1) {
					e1.printStackTrace();
				}


			}


});
		lblFachhochchule.setLayoutData(new GridData(SWT.CENTER, SWT.CENTER, false, false, 1,
1));
		lblFachhochchule.setText("Fachhochschule D\u00FCsseldorf\r\nLabor f\u00FCr Informatik
und Embedded Systeme\r\nTel: 0211 4351 - 347\r\nE-Mail: aal@inflab.et.fh-duesseldorf.de");
		lblFachhochchule.setAlignment(SWT.CENTER);


		//createPartControl(parent);
	}

	@Focus
	public void setFocus() {
		// Set the focus
	}

}
```